Git in Four Weeks - Week 2
Revision 1.1
06/09/21

Note: This series of labs assumes the first 3 labs from week 1 have been completed and you start out working in that same directory. If you don't have that directory, you can use the git repo as shown below. (Note you only need to do these steps if you don't have your repo from last week.)

```
$ git clone https://github.com/skillrepos/git4-2
$ cd git4-2
$ git remote rm origin
```

Lab 4 - Working with Branches

In this lab, we'll start working with branches by creating a new branch and making changes on it.

Steps

1.) Starting in the same directory as you used for Lab 3, take a look at what branches you have currently with the git branch command.

```
$ git branch
```

2.) You'll see a line that says "* master". This indicates that there is only one branch currently in your repository - master. The "*" next to it indicates that it is the current branch (the one you've switched to and are currently working in). If your terminal prompt is configured to show the current branch, it would also say "master".

3.) Now, before we work with a new branch, let's update the files in the master branch to indicate that these are the versions on master so it will be easier to see which version we have later. To do this, we can just use a short version of the same way we have been creating and updating other files.

On some non-Windows systems, run this command.

```
$ echo "master version" >> *
```

On Windows and some other systems, it will be necessary to issue the command for each file, such as:

```
$ echo "master version" >> file1-name
$ echo "master version" >> file2-name
…
```

4.) Stage and commit the updated files. Because these are files that Git already knows about, we can use the shortcut command here.

```
$ git commit -am "master version"
```

5.) When we work with branches, it can be helpful to see a visual representation of what's in the repository. To do this, we'll use the Gitk tool that comes with Git. Start up gitk in this directory and have it run in the background.

```
$ gitk &
```

6.) In gitk, create a new view. Follow the instructions below.

1. **On the menu, select View, New View. Give it a name.**
2. **Check "Remember this view"**
3. **Check the four checkboxes under the "Branches & tags:" field.**
4. **Click OK.**
5. **Check that the new view is now selected under the View menu. If not, select new view under that menu.**

7.) We have a new feature to work on, create a branch for the feature. Switch back to your terminal, and in the directory, run the command below.

```
$ git branch feature-branch-name
```

8.) Notice that this command created the branch but did not switch to it. Let's check what branches we have and which is our current one.

```
$ git branch
```

9.) We can now see our new branch listed. Let's change into the feature branch to do some work.

```
$ git checkout feature-branch-name
```

10.) Verify that we're on the feature branch. Run the command below and observe that the "*" is next to that branch.

```
$ git branch
```

11.) Switch back to gitk, and refresh the screen to see what things look like visually now. (Note, to refresh you may need to use the "Reload" or "Update" function under the "File" menu in gitk.)

12.) Back in the terminal session, first create a new file. Then update the files in the feature branch to indicate that they are the "feature branch version".

```
$ echo "some-text" > new-file
```

Next, update the files with a way to indicate these are the version of files on the new branch. On some non-Windows systems, you can use:

```
$ echo "feature version" >> *
```

On other systems, it will be necessary to issue the command for each file, such as:

```
$ echo "feature version" >> file1-name
$ echo "feature version" >> file2-name
…
```

13.) When you're done, stage and commit your changes.

```
$ git add .
$ git commit -m "feature version"
```

(Note: If you just used git commit -am, it wouldn't pick up your new file.)

14.) Refresh your view in gitk and take one more look around your feature branch.

15.) Switch back to the master branch.

```
$ git checkout master
```

16.) Verify you're on the right branch.

```
$ git branch
```

(Note: Should have a * by master.)

17.) Take a look at the contents of the files and verify that they're the original ones from master.

$ cat * (or "type *" on Windows)

Look for "master version" in the text.

Lab 5 - Practice with Merging

In this lab, we'll work through some simple branch merging.

Prerequisites

This lab assumes that you have done Lab 4: Working with branches. You should start out in the same directory as that lab.

Steps

1.) Starting in the same directory as you used for Lab 4, make sure you don't have any outstanding or modified files (nothing to commit). You can do this by

running the status command and verifying that it reports “working directory clean”.

```
$ git status
```

2.) Now, create a new one-line file with a line that identifies it as the master version.

```
$ echo "some-text" > file5-name
```

3.) Stage it and commit on the master branch.

```
$ git add .
$ git commit -m "adding new file on master"
```

4.) Start up gitk if it’s not already running.

```
$ gitk &
```

5.) Create a new branch but don't switch to it yet. (You can use whatever branch name you want.)

```
$ git branch new-branch
```

6.) Change the same line in the new file (still on master)

```
$ echo "Update on master" > file5-name
```

7.) Stage and commit that change (still on master)

```
$ git add .
$ git commit -m "update on master"
```

8.) Switch to your new branch.

```
$ git checkout new-branch
```

9.) Now on newbranch, make a change to the same line of the same file.

```
$ echo "Update on new-branch" > file5-name
```

10.) Stage and commit it on newbranch.

```
$ git commit -am "update on new-branch"
```

11.) Switch back to the master branch.

```
$ git checkout master
```

12.) Merge your new branch back into master. (This will attempt to merge newbranch into master.)

```
$ git merge new-branch
```

13.) Check the status of things. Notice that we have a conflict.

```
$ git status
```

14.) Also take a look at the local file and notice the conflict markers.

```
$ cat file5-name (or "type file5-name" on Windows)
```

15.) "Fix" the conflict in the file in the working directory. (For simplicity you can just write over it.)

$ echo "merged version" > *file5-name*

16.) Stage and commit the fixed file.

$ git commit -am "Fixed conflicts"

17.) Check the status to make sure the merge issue is resolved

$ git status

18.) Refresh gitk and take a look at the changes.

19.) We're done with your new branch, so get rid of the branch.

$ git branch -d *new-branch*

20.) Refresh gitk and take a look at the most recent changes.

Lab 6 - Using the Overall Workflow with a Remote Repository

In this lab, you'll get some practice with remotes by working with a Github account, forking a repository, cloning it down to your system to work with, rebasing changes, and dealing with conflicts at push time.

Prerequisites

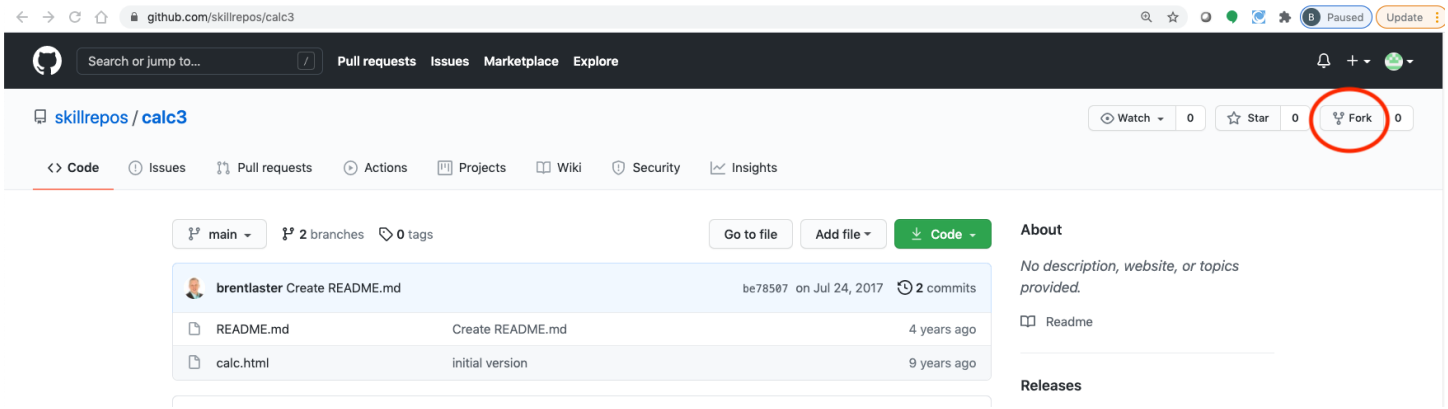
This lab assumes that you have internet access.

Steps

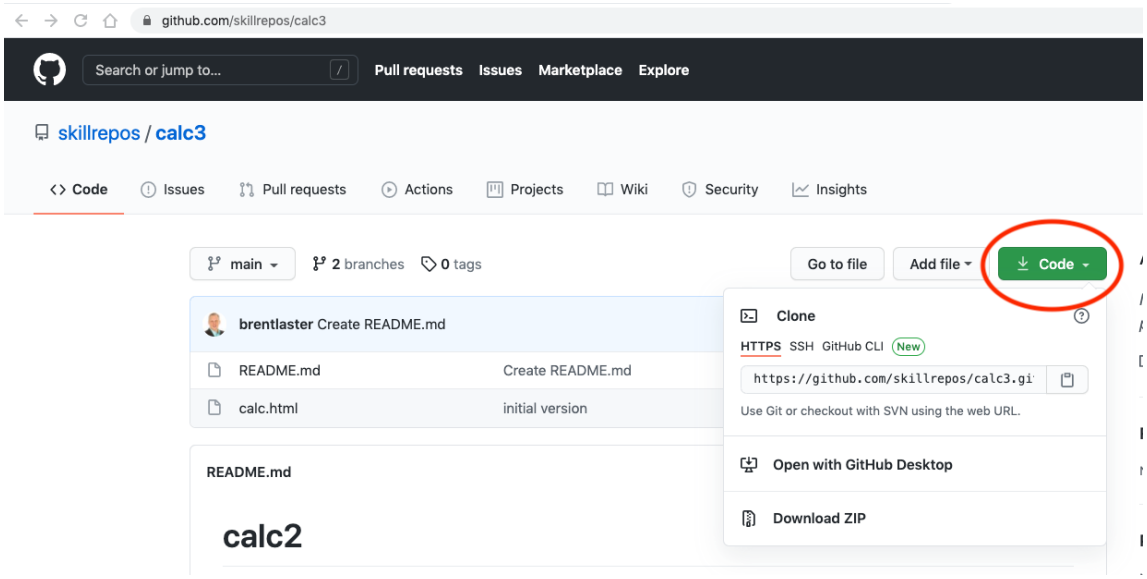
1.) Go to **https://github.com** and sign in to your Github account.

2.) Browse to the calc3 project at https://github.com/skillrepos/calc3

3.) Click on the **Fork** button (top right) and the repository will be forked to your userid. (Your URL should change to https://github.com/*your-github-userid*/calc3.)



4.) On the GitHub screen, near the middle right, should be a green button, labelled "Code". Click on that. A new window pops up populated with the URL path you can use to clone this project down via the https protocol. To the right of that command, you'll see a "clipboard" icon. Click on that to copy the path to your clipboard. This saves you from having to construct the path yourself.



5.) Switch back to your terminal session. CD back up a level if needed to make sure you are not in one of the existing projects from the other labs.

Then clone the project down by typing “git clone” and then pasting the path from the clipboard. Hit Enter.

$ cd .. (if needed)
$ git clone https://github.com/*your-github-userid*/calc3.git

6.) You should see some messages from the remote side and then the project will be cloned down into the calc3 directory.

Change (**cd**) into the calc3 directory.

$ cd calc3

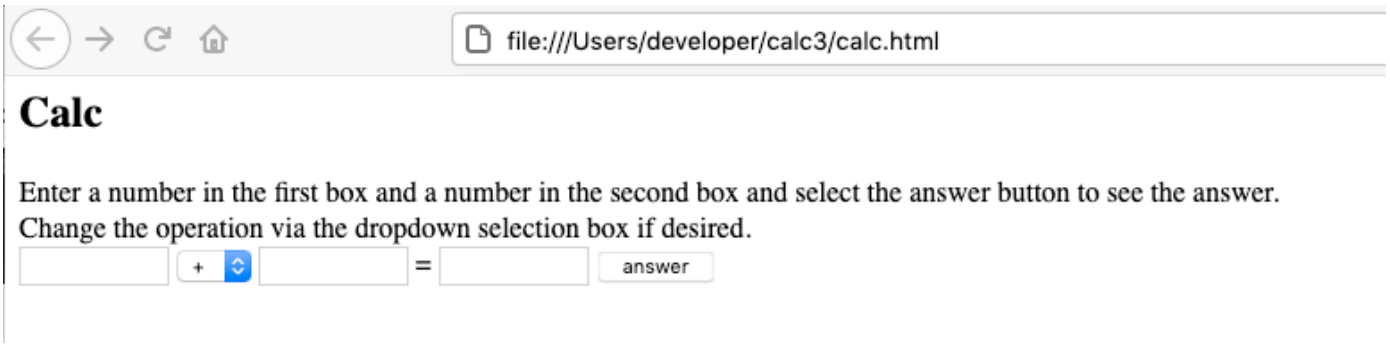
7.) You can now browse around the calc3 directory. There are only two files in there, but if you look at the hidden files, you’ll see the **.git** repository that was cloned down from the remote. You can also run commands like branch against them to see the set of branches. Also try the commands below to see the list of remote branches and information about the most recent set of changes in each.

$ git branch -r
$ git branch -av

8.) You can then run the remote -v operation to see the remote.

$ git remote -v

9.) (Optional) Let’s see what features our calculator already has. Open up the **calc.html** program in a browser and take a look. You’ll notice we have the basic arithmetic functions there: addition, subtraction, multiplication, and division.



10.) We want to incorporate some other features into our calculator program from the features branch. First, we'll setup a local features branch to track where the remote branch is in the repo we just cloned. Create a local branch tracking the features remote branch.

```
$ git branch features origin/features
```

11.) Let's get a look at what's in the main branch and what features are available for us to use in the features branch.

```
$ git log --oneline          (to see what's on main)
$ git log --oneline features  (to see what's on features)
```

12.) We want to merge in the max, exp, and min functions to add to our calculator. We'll do this with a rebase to get the history as well. Still in your main branch, run the rebase command as follows.

```
$ git rebase features
```

13.) Once that finishes, you can do a quick log of your current branch (main) and see that the history records show up there now.

```
$ git log --oneline
```

(Optional) You can also open up the calc.html program in a browser and verify that the functions are there as well.

14.) Assuming the rebase was successful, push the updates out to the remote.

```
$ git push origin main
```

15.) At this point, you'll see an error message about Git "rejecting" your push due to a "non-fast-forward" situation. Basically, it can't do a fast-forward merge. To correct this, we need to merge in the "more up-to-date" content from the remote. To try this, we can just do a pull operation from the remote.

```
$ git pull
```

Git will want to create a new "merge commit" for this and will prompt you for a commit message (via the editor). You can change it if you want, but when you are done, close the editor.

16.) In this case, the merge was fairly simple and should have succeeded. Now that we are up-to-date, we can try the push again. This time it should succeed without problems.

```
$ git push origin main
```

17.) If you want, you can look back in GitHub and see the update from the merge.